2-tier vs. 3-tier Architectures for Data Processing Software

Dmitriy Dorofeev
YASP Ltd.
Saint-Petersburg, Russia
dima@yasp.com

Sergey Shestakov
Luxms Group
Saint-Petersburg, Russia
serg@luxmsbi.com

ABSTRACT

The rise of data-centric computing, NoSQL and newSQL databases with powerful scripting capabilities, popularity of REST API raise a question: is it feasible to serve clients directly from the DB, with REST API server residing inside the database? What would be the balance between data processing, application, and presentation logic for such a scenario on a server side and on a client side?

In this paper we will compare 2 real-world implementations of the commercial Luxms BI analytical platform based on 2-tier and on 3-tier architecture.

Our research shows that despite popularity of 3-tier architectures, in-database application server approach delivers better performance in both throughput and latency in analytical client-server application development.

CCS CONCEPTS

• **Software and its engineering → n-tier architectures**; **3-tier architectures**;

KEYWORDS

software architecture, 2-tier, 3-tier, benchmarks, data processing, data centric

ACM Reference Format:
Dmitriy Dorofeev and Sergey Shestakov. 2018. 2-tier vs. 3-tier Architectures for Data Processing Software. In *The 3rd International Conference on Applications in Information Technology (ICAIT'18), November 1–3, 2018, Aizu-Wakamatsu, Japan.* ACM, New York, NY, USA, Article 13, 6 pages. https://doi.org/10.1145/3274856.3274869

1 INTRODUCTION

Traditionally, 3-tier architectures were the most popular among all possible multi-tier (n-tier) architectures. In a 3-tier architecture, there are separate subsystems for presentation, business/application logic, and data management [9]. With 2-tier architectures, it's presentation layer running on a client and a data management layer residing on a server. Application logic may reside on the client (fat clinet) or on the server (thin client). Some researchers encouraged data centric design for n-tier architectures to provide better performance and faster development [4] [8].

ICAIT'18, November 1–3, 2018, Aizu-Wakamatsu, Japan

ACM ISBN 978-1-4503-6516-1/18/11…$15.00
https://doi.org/10.1145/3274856.3274869

Data-centric approach becomes even more important for enterprise analytical processing. Recent explosive growth of data (internet of things, social networks, etc.) shifts value from business logic to data itself [5] and requires re-architecturing of current solutions for the big data world.

Google implemented MapReduce algorithm in the distributed environment on a large cluster of commodity machines [2] clearly follow data-centric architecture.

Another promising trend in data-centric architectures is fog computing [1], with application logic placed on fog servers where data are pre-collected and pre-aggregated before going to the long term Cloud storage and analytics.

If we consider 2-tier architecture with a fat client, the big data challenge makes network communications a bottleneck for any analytical task, though there are attempts to address that issue, like RIA cloud architectures [11] [12]. Rather, implementation of a smart thin client designed for minimizing network traffic and minimizing client side data cache should address the big data problem in a more efficient way. Such a client should be able to delegate heavy, data-intensive queries to the server leveraging Massive Parallel Processing/Hadoop storages, and receive chunks of data optimized for visualization.

Strong efforts were made to provide in-memory computing capabilities [7]. Though very robust and quite successful for certain types of workloads, non-relational in-memory engines (NoSQL including) proved not that flexible and functionally powerful for data-intensive analytical processing and were not able to replace SQL-based logic. It proved very inefficient to replace complex SQL queries with manually written business logic composed of numerous loops and conditional statements.

In short, original NoSQL engines failed to provide developers of analytical systems with tools matching relational algebra. Developers pushed many NoSQL vendors to implement SQL support, examples include Cassandra, EllasticSearch and CouchBase. We see more and more examples of hybrid NewSQL engines [10], which try to combine the best of two worlds - ACID properties (Atomicity, Consistency, Isolation, Durability) and relational algebra running against distributed databases with in-memory support [6]. Still these engines lack the advanced query power of classical relational database management systems (RDBMS), their support of SQL mostly is limited to some subset of ANSI SQL 99, and lacks procedural extensions.

In the meanwhile, RDBMS evolved its querying and in-memory capabilities. Modern relational database engines now offer more and more advanced capabilities of procedural language extensions to SQL, which are good enough to implement all the required business logic inside the database. This creates new opportunities for implementing data-centric architectures which is particularly important for such data-intensive tasks as analytical processing.

2 DATA-CENTRIC APPROACH FOR ANALYTICAL PROCESSING

Analytical processing tasks frequently require running many queries to the database for completion of a single user interaction. This is especially true for analytical data visualization. For example, even preparing a single dashboard requires lots of requests and interactions with data sources, and even running these in parallel won't solve the problem completely. 3-tier architecture allows to run queries in parallel, yet requires a lot of efforts to sync the processing of the results.

Obviously data transfer between database and application server will be a bottleneck for such applications. Any optimization in this area will improve latency and overall response times.

All these considerations along with recent advancements of RDBMS led us to a concept of data-centric 2-tier architecture, with in-database application server.

Our implementation of 2-tier architecture is based on PostgreSQL 9.6, with application logic coded in PL/pgSQL. PL/pgSQL is most suitable for data-centric applications because it has excellent SQL integration, JSON processing support, rich library of text and math functions.

The suggested database-centric 2-tier approach is not PostgreSQL specific, it may be implemented in any database with stored procedures support. For example: Oracle commercial RDBMS, Tarantool NoSQL database and Redis in-memory key-value storage.

3 3-TIER ARCHITECTURE REFERENCE IMPLEMENTATION

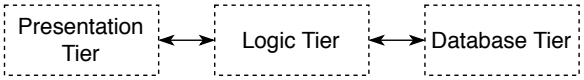

Figure 1: 3-tier architecture

Our 3-tier implementation consists of a client browser based application implemented in HTML 5/Javascript, application server implemented in Java 8, and a database server implemented in PostgreSQL 9.6. We run Nginx as an HTTP proxy in front of an application server to provide security and load balancing.

Let's consider data flow from the client to the server and back.

(1) Client creates a JSON formatted HTTP request for the data
(2) HTTP proxy forwards the HTTP request to the application server as-is
(3) Application server parses the request and creates a series of SQL queries based on this
(4) JDBC layer (part of the application server) uses a wire database protocol to send the SQL queries to the database

After the completion of database querying, the data is flowing back in reverse order. The overall process is illustrated in Figure 2.

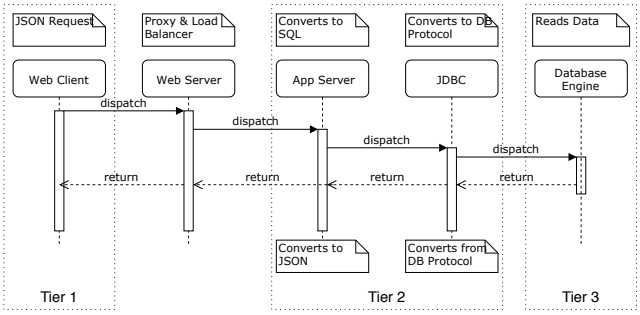

Figure 2: 3-tier request processing diagram

In practice, application server needs to make several queries to the database in order to process incoming request and assemble the response. So we can consider interaction with database as a potential bottleneck for the 3-tier implementation, even if SQL queries will run in parallel within one client request.

4 DATA-CENTRIC 2-TIER ARCHITECTURE REFERENCE IMPLEMENTATION

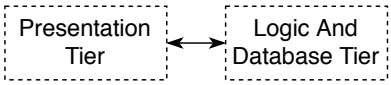

Figure 3: 2-tier architecture

Our data-centric 2-tier implementation consists of the same client browser based application implemented in HTML 5/Javascript and an in-database application server implemented in PL/pgSQL along with data schema located within the same PostgreSQL 9.6 database engine. Nginx plays a role of HTTP proxy handling requests in front of the application server to provide security and static files serving. We've used HAProxy for load balancing and queueing incoming queries to the application server.

Here's a client-server data flow described step-by-step:

(1) Client creates a JSON-formatted HTTP request for the data (format is identical to the 3-tier client data format)
(2) HTTP proxy forwards the HTTP request to the in-database application server calling a stored procedure with HTTP request headers and body as parameters
(3) In-database application server parses request and dispatches it to the appropriate PL/pgSQL procedure. As a result, several SQL queries can be executed by the database based on business logic and request details.

Similarly to the 3-tier architecture described in Section 3, the data upon querying completion starts to flow back in reverse order. This process is shown in Figure 4.

Nginx functionality can be extended with Lua programming language, so we've implemented a proxy between HTTP and stored procedure - this required about 200 lines of Lua code.

In-database application server is written in PL/pgSQL and implements HTTP routing, authentication, authorization, generic CRUD operations on tables, and specific API end-points when CRUD is not

enough. SQL queries and PL/pgSQL procedures cannot run in parallel within one client session, a significant drawback as compared with Java-based 3-tier implementation.

The complete application server codebase is roughly 6000 PL/pgSQL lines of code (LOC). This is verbose implementation of application logic as there are no third-party modules typical for Java ecosystem. The HTTP request routing part is only 800 LOC.

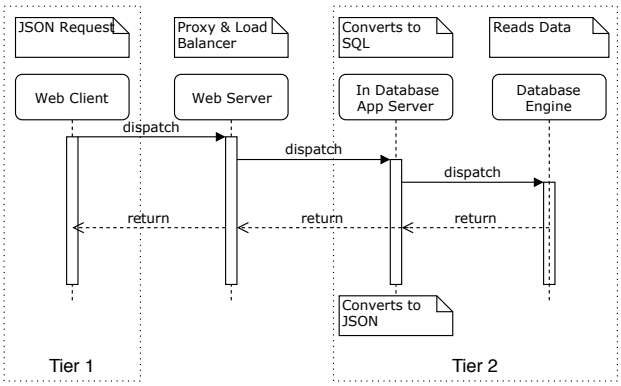

Figure 4: 2-tier request processing diagram

5 2-TIER VS 3-TIER SOFTWARE DEVELOPMENT LIFE CYCLE COMPARISON

In this section we outline pros and cons of the tools available for Software Development Life Cycle (SDLC) phases of our 3-tier and database-centric 2-tier reference implementations, respectively. Historically, shift of popularity from 2-tier to 3-tier architectures was heavily influenced by better tools for SDLC, so it's important to revisit and check current state of the tools available.

5.1 Development

Java-based 3-tier posesses fully-functional, feature-rich IDE, debugger and profiler. PL/pgSQL-based data-centric 2-tier does not provide fully-fledged IDE, debugging is possible with tools like omniDB/pgAdmin, profiling is possible with PL Profiler extension or pg_stat_statements view. Thus, Java offers better development tools, but PL/pgSQL provides a must-have set of tools for software development.

Java is a compiled language and requires to build complete application before it can be run with JVM. PL/pgSQL allows to change code on a per function granularity, significantly faster than with Java.

5.2 Testing

For testing, Java-based 3-tier has wide selection of frameworks and tools for every taste. PL/pgSQL-based data-centric 2-tier presumes that developer should take care of testing environment, and only few frameworks are available. It's easier to setup test environment with Java, but it can be done with PL/pgSQL as well.

5.3 Deployment

Java-based 3-tier provides good support for JAR file upload & restart, even hot reload with Java Servlet Containers is supported. PL/pgSQL-based data-centric 2-tier has decent support for transactional batched CREATE OR REPLACE FUNCTION scripts. It's similar to hot reload, but requires some tooling and discipline. So, for Deployment phase, PL/pgSQL matches Java.

5.4 Operation

For runtime, classical 3-tier requires an additional process and/or server with JVM settings tuning. Data-centric 2-tier runs in the database process, no extra hardware is required.

For monitoring, classical 3-tier can be monitored using wide selection of JMX-compatible tools, whilst data-centric 2-tier can be monitored with standard database tools. Thus, for Operational phase, classical 3-tier solution is better monitored, but data-centric 2-tier is up to the task too.

5.5 Maintenance and Support

It is resource-consuming to keep the application server of classical 3-tier on par with database schema. Flawless upgrade may be tricky and require interruptions of production. While minor changes can be done easily, undertaking some major changes lead to very complex and risky projects. For data-centric 2-tier, on opposite, it's easy to keep in sync the application server and database schema. Minor changes can be done quickly, and major changes can be done with due preparations. But this flexibility should come in hand with discipline and strict rules. We come to a conclusion that maintaining and Supporting phase for a 3-tier implementations is much more complex as compared with flexible data-centric 2-tier.

6 BENCHMARKING

Benchmarking was performed with Locust [3] open source tool. Locust was configured to run 1 master and 4 slave processes on a dedicated Centos 7.2.1511 Linux virtual machine with 16 CPUs and 8Gb RAM. Locust was configured to send HTTP requests randomly with minimum delay between requests of 1 seconds and maximum delay of 3 seconds per Locust virtual user.

We were running Luxms BI server on a Centos Linux version 7.3.1611, dedicated virtual machine with 8 CPUs and 16Gb RAM. 3-tier and 2-tier versions were running on the same hardware and leveraged PostgreSQL 9.6.5 database. In order to check different aspects of server-side implementations we've generated several types of workload.

6.1 Lightweight workload

Lightweight workloads were run to measure maximum possible throughput using specially designed random requests - the idea was to generate tiny responses, but still require server to perform some SQL activity. 4000 Locust users were configured to generate the load. In total, 200,000 lightweight requests were measured. As a result, 2-tier solution has demonstrated 3 times lower latency (Figure 5), and processed twice more requests per second (Figure 6).

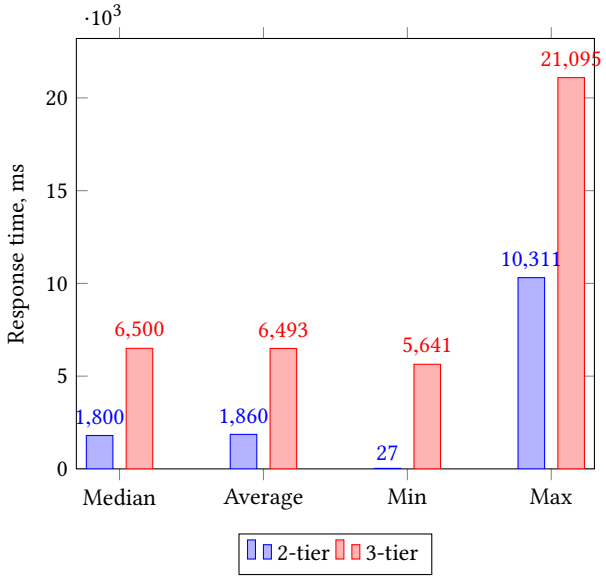

Figure 5: 2-tier vs. 3-tier response time statistics (lightweight workload)

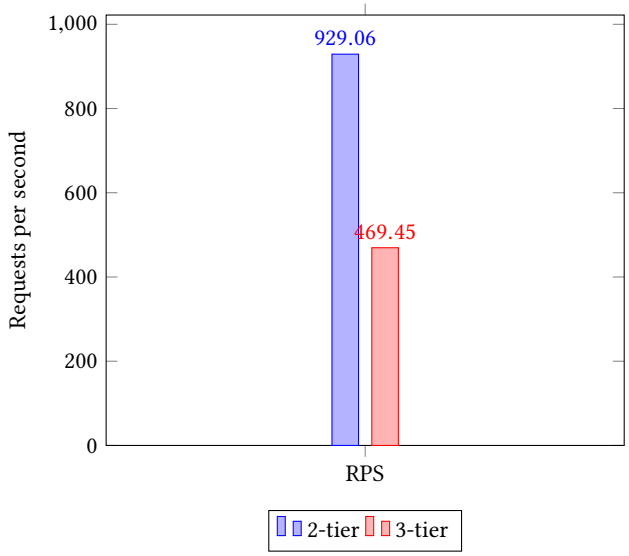

Figure 6: 2-tier vs. 3-tier RPS (lightweight workload)

Distribution of response times under lightweight workload is presented on Figure 7.

3-tier implementation under workload of 5,000 Locust users resulted in massive timeouts, while 2-tier performed well. We conclude that 5,000 simultaneous users is the upper limit for 3-tier implementation in our test environment.

6.2 Medium Workload

Medium workloads were run using real-world dataset with about 90 millions records in facts table. Running random requests against

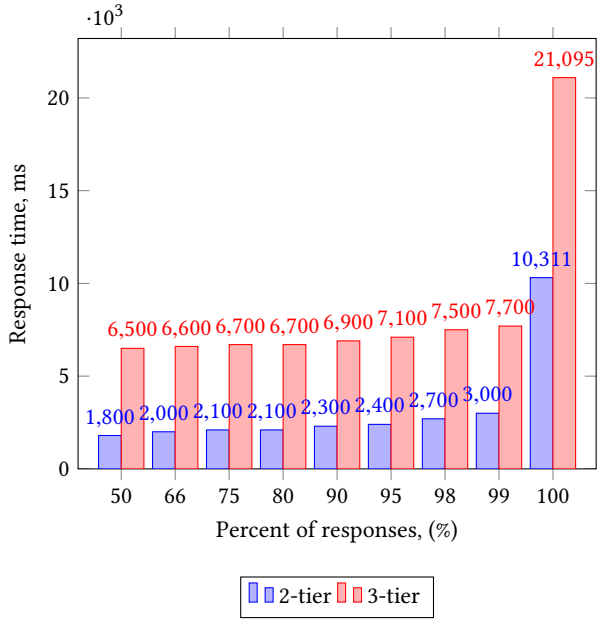

Figure 7: 2-tier vs. 3-tier response time - percentile (lightweight workload)

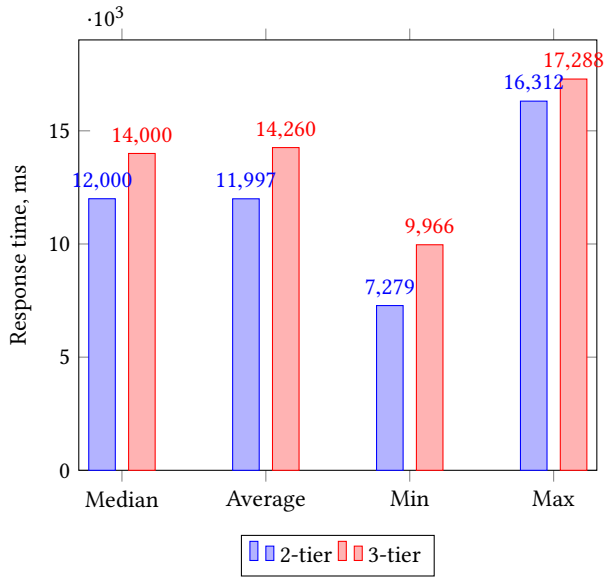

Figure 8: 2-tier vs. 3-tier response time statistics (medium workload)

this dataset resulted in responses sized up to 120Kb in JSON format. Server needed to check 3 tables with dimensions data and scan facts table randomly to prepare responses. Typical SQL query against facts table took 100ms on a database side (direct SQL query), so it is absolute minimum response time for the application server.

400 Locust users were configured to perform these requests.

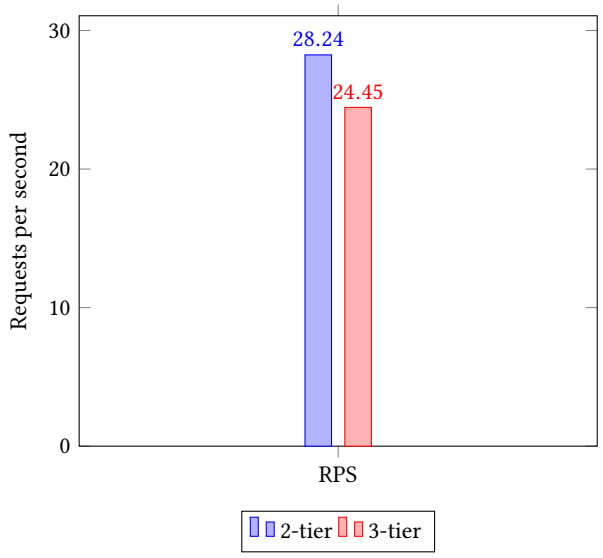

Figure 9: 2-tier vs. 3-tier RPS (medium workload)

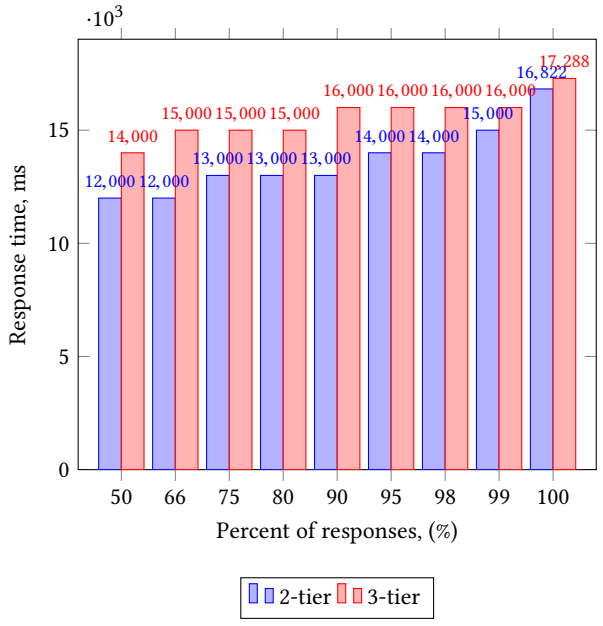

Figure 10: 2-tier vs. 3-tier response time - percentile (medium workload)

According to our benchmarks, 2-tier implementation performed 15% better on Medium workloads (Figures 8, 9 and 10).

6.3 High Workload

High workload leverages same queries as Medium one, but with increased number of simultaneous users, up to the server limit.

With 800 Locust users, classical 3-tier was able to complete only 1875 requests and all other requests were failing. Because of that we provide statistics only for 2-tier, which has only 19 timeout errors out of 10,000 requests.

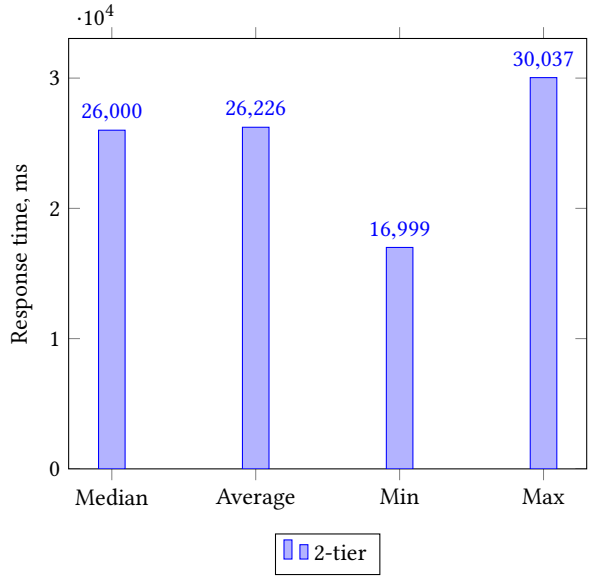

Figure 11: 2-tier response time statistics (high workload)

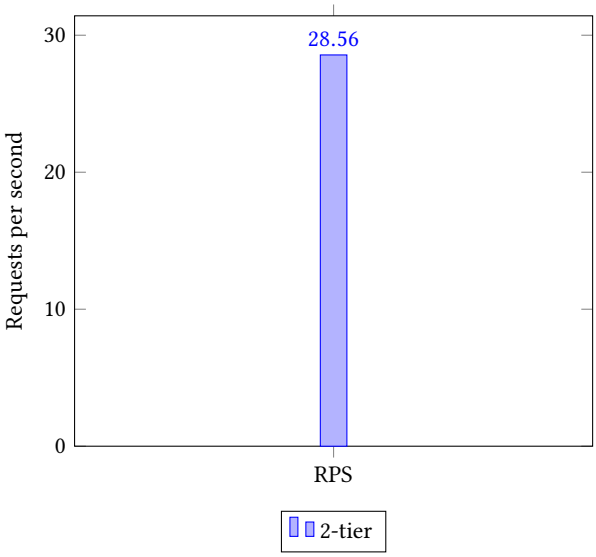

Figure 12: 2-tier RPS (high workload)

2-tier response time statistics for heavy workload is shown on Figure 11 and measured RPS is on Figure 12.

7 CONCLUSIONS

We have compared and benchmarked implementations of a classical 3-tier architecture and data-centric 2-tier architecture for Luxms BI, an analytical engine for decision making.

Our benchmarks demonstrated that data-centric 2-tier architecture with in-database app server has 15% better performance as

compared with classical 3-tier architecture with Java app server on identical hardware resources. On synthetic benchmarks 2-tier implementation shown a twice better RPS results than 3-tier one.

The data-centric approach also brings development more agility with features like incremental code updates.

More detailed analysis of the gathered results is a subject for future work. Extensive profiling is required to find slowest parts of the query processing pipeline for both 2-tier and 3-tier implementations.

To sum up, the data-centric 2-tier architecture implementation is more robust, resource efficient and agile, as compared with classical 3-tier implementation.

REFERENCES

[1] Rabindra K. Barik, Harishchandra Dubey, and Kunal Mankodiya. 2017. SoA-Fog: Secure Service-Oriented Edge Computing Architecture for Smart Health Big Data Analytics. *CoRR* abs/1712.09098 (2017). arXiv:1712.09098 http://arxiv.org/abs/1712.09098

[2] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150.

[3] Jonatan Heyman, Carl Byström, Joakim Hamrén, and Hugo Heyman. 2017. Locust. Retrieved June 11, 2018 from https://locust.io/

[4] Paul D. Manuel and Jarallah AlGhamdi. 2003. A Data-centric Design for N-tier Architecture. *Inf. Sci. Inf. Comput. Sci.* 150, 3-4 (April 2003), 195–206. https://doi.org/10.1016/S0020-0255(02)00377-8

[5] Timothy Prickett Morgan. 2016. The Emergence Of Data-Centric Computing. Retrieved June 11, 2018 from https://www.nextplatform.com/2016/10/06/emergence-data-centric-computing/

[6] John Piekos. 2015. SQL vs. NoSQL vs. NewSQL: Finding the Right Solution. Retrieved June 11, 2018 from http://dataconomy.com/2015/08/sql-vs-nosql-vs-newsql-finding-the-right-solution/

[7] P. Siegl, R. Buchty, and M. Berekovic. 2016. Data-Centric Computing Frontiers: A Survey On Processing-In-Memory. In *Proceedings of the Second International Symposium on Memory Systems, MEMSYS 2016, Washington, DC, USA, October 3-6, 2016*. ACM, 295–308. https://doi.org/10.1145/2989081.2989087

[8] Nitin Uplekar. 2001. Building Data-Centric n-Tier Enterprise Systems. Retrieved June 11, 2018 from http://www.powervision.com/html/news/n_tier_arch.pdf

[9] Wikipedia. 2018. Multitier architecture — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Multitier%20architecture&oldid=822900859. [Online; accessed 11-June-2018].

[10] Wikipedia. 2018. NewSQL — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=NewSQL&oldid=837852262. [Online; accessed 13-June-2018].

[11] Wenjun Zhang. 2010. 2-Tier Cloud Architecture with maximized RIA and SimpleDB via minimized REST. *2010 2nd International Conference on Computer Engineering and Technology* 6 (2010), V6–52–V6–56.

[12] Wenjun Zhang. 2012. 2-Tier Cloud Architecture and Application in Electronic Health Record. *JSW* 7 (2012), 765–772.